Predicting and Controlling Resource Usage in a Heterogeneous Active Network

V. Galtier, K. Mills, Y. Carlinet
National Institute of Standards and Technology
Gaithersburg, MD 20899-8920
{kevin.mills,virginie.galtier}@nist.gov

S. Bush, A. Kulkarni
General Electric
Corporate Research & Development
KWC-512, One Research Circle,
Niskayuna, NY 12309
{bushsf,kulkarni}@crd.ge.com

Abstract

Active network technology envisions deployment of virtual execution environments within network elements, such as switches and routers. As a result, inhomogeneous processing can be applied to network traffic. To use such technology safely and efficiently, individual nodes must provide mechanisms to enforce resource limits. This implies that each node must understand the varying resource requirements for specific network traffic. This paper presents an approach to model the CPU time requirements of active applications in a form that can be interpreted among heterogeneous nodes. Further, the paper demonstrates how this approach can be used successfully to control resources consumed at an active-network node and to predict load among nodes in an active network, when integrated within the Active Virtual Network Management Prediction system.

1. Introduction

Internet applications increasingly use mobile code, such as applets, servlets, scripts, and dynamically linked libraries, to deliver new software to millions of users. Without understanding the processing (CPU time) required by such dynamically downloaded software, computer operating systems cannot effectively manage system resources or control the execution of mobile code. Unfortunately, since mobile code can be downloaded and executed on a wide variety of computer systems with a vast range of capabilities, software developers cannot specify CPU-time requirements a priori. This problem exists because CPU-time requirements do not depend solely on the processor speed of various computers, but rather on a complex array of hardware and software factors.

We investigated how to solve this problem in the context of a heterogeneous active network (heterogeneity implies that the active network comprises a wide range of node types with various hardware capabilities and software configurations). In Section 2, we provide a brief tutorial on active networks, and we summarize the factors that can cause CPU usage to vary as an active application moves among heterogeneous nodes in an active network. In Section 3 we motivate the need for a method to express meaningful processing requirements for active applications. Section 4 briefly reviews existing solutions and their limitations. Section 5 outlines our approach to model and predict CPU usage, and introduces new measurements confirming our earlier results [7]. Subsequently, we exercise this technique in two experiments involving active-network applications [19]. In the first experiment, described in Section 6, we compare the effectiveness of controlling CPU usage in active-network nodes using two different policies: (1) assigning a fixed CPU limit per packet independent of node characteristics and (2) assigning CPU time per packet adjusted to account for differences among nodes. In a second experiment, we incorporate a CPU-time prediction model into the Active Virtual Network Management Prediction (AVNMP) system [4], a technology developed by researchers at General Electric (GE) to predict load in a network. Section 7 discusses AVNMP, and then in Section 8 we explain how we augment AVNMP to predict CPU usage in heterogeneous active networks. We also outline an experiment where we use AVNMP to predict resource consumption by an active-audio application. Section 9 presents a critical view of our approach and introduces ideas we would like to explore to improve our work. We present our conclusions in Section 10.

2. Active network nodes, heterogeneity, and resource variability

Active-network technology augments traditional networking with the possibility that individual packets carry executable code, or references to executable code. Each

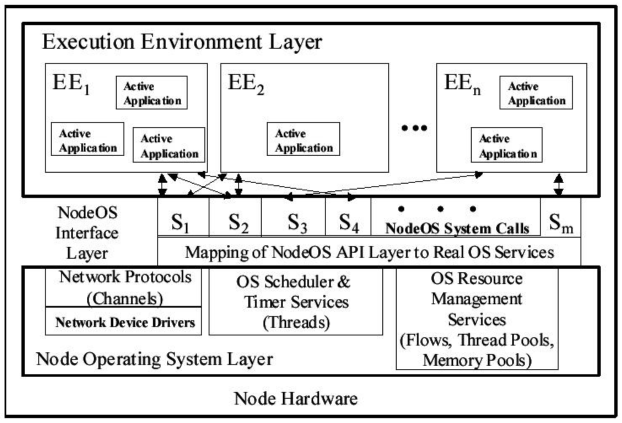
Execution Environment Layer
EE$_1$
Active Application
Active Application
Active Application
EE$_2$
Active Application
EE$_n$
Active Application
Active Application
NodeOS Interface Layer
S$_1$
S$_2$
S$_3$
S$_4$
NodeOS System Calls
S$_m$
Mapping of NodeOS API Layer to Real OS Services
Network Protocols (Channels)
Network Device Drivers
OS Scheduler & Timer Services (Threads)
OS Resource Management Services (Flows, Thread Pools, Memory Pools)
Node Operating System Layer
Node Hardware

Figure 1. Architecture of an active network node

router forwards conventional packets on a fast path, while delivering active packets to a higher-level execution environment (EE) that can identify and run code associated with the packet. Networking applications built with active packets are referred to as active applications (AAs). Figure 1 illustrates the architecture of an active-network node [5].

Underlying each active-network node is a node operating system, which transforms the hardware into a software abstraction to provide EEs with controlled access to resources such as CPU cycles, memory, input and output channels, and timers. To permit various operating systems to provide services to various EEs, the node architecture includes a standard specification for system calls (the Node OS Interface Layer in Figure 1) [14]. Each EE accepts active packets that initiate execution of packet-specific code.

Analysis of Figure 1, and real systems, reveals many sources of variability affecting CPU usage by AAs [6]. In the hardware layer, the main factors include: processor architecture and frequency, available memory, and the speed of various internal buses and network and disk interfaces. Within the node operating system (OS) and node OS interface layer, the factors include: performance of device drivers, performance in managing processes and memory, and the nature and performance of system calls provided by the operating system. For networking system calls, performance of reads and writes varies based on the specific protocol stacks buried beneath the systems calls, as well as the implementation of those protocols. Within the EE layer, performance can be affected by mapping EE system calls onto OS system calls, as well as by the compiler and options used to compile the EE. Finally, each execution of a specific AA can go through any one of many program paths. The path of any given execution can depend on data in the packet, on the state of the node, and even the state of other nodes. To properly estimate CPU-time requirements for a given AA on a specific node, we need a model that accounts for these sources of variability.

3. Why is it necessary to model the CPU requirements of active packets?

In classical networks, each packet is treated the same; thus, the CPU time required to process a packet includes a fixed per-packet overhead and a per-byte overhead. Using this knowledge, designers of an intermediate node can quite easily estimate the CPU time needed to process a packet, and can thus rate the capacity of the node in packets per second. Since packets in an active network can include code describing how to treat the packet, the CPU time needed to process each active packet can vary significantly, and can be difficult to estimate.

Inability to estimate the CPU time required by an active packet can lead to some significant problems. First, an active packet might consume excessive CPU time at a node, causing the node to deny services to other packets (see related experiment in Section 6). Such excessive CPU time consumption might be due to maliciously or erroneously programmed code carried by an active packet. Second, an active node may be unable to schedule CPU resources to meet the performance requirements of packets. Third, an active packet may be unable to discover a path that can meet its performance requirements. This path selection problem occurs in part due to the node-scheduling problem, but also because the CPU time commitments of active nodes along a path cannot be determined (see related experiment in Section 8).

Devising a method for active packets to express their CPU time requirements can help to resolve these problems, and can open up some new areas of research. For instance, Qie et al [15] explore the design space for scheduling the CPU on software-based routers. This work relies on the assertion that the cycle rate required by QoS (quality of service) flows can be derived empirically from a specified bit rate. However, our experience shows that in active networks the number of cycles required to process each packet can vary greatly from one node to another. A model such as we propose could help Qie and colleagues to develop a signaling protocol by which an application reserves a particular cycle rate. Elsewhere, RCANE (Alexander et al [1]) allows a user to reserve a guaranteed allocation of CPU, denoted as a slice of CPU received over a period of time. For example, if an application processes a network video stream where packets arrive every 20 ms and require 0.5 ms to process, then the application should request a reservation for a 0.5 ms slice of CPU time every 20 ms of real time. This approach supposes that an application knows how much CPU time is required to execute its packets on each and every node. A model such as we propose might provide the necessary estimates.

Even though we concentrate on active networking and demonstrate our ideas in this context, our approach might also apply more generally for example to heterogeneous distributed systems, where processes execute on nodes that exhibit a wide range of computational capabilities. Our approach should apply particularly to heterogeneous distributed systems based on mobile code, such as mobile agent applications, and also to systems based on code loaded from disk, such as distributed parallel processors or web-based applications.

4. Related work

In this section we present existing solutions to prevent excessive CPU resource consumption in active networks, and we describe the limits of these solutions. Next we examine research conducted outside of active networks that could help to solve our problem.

4.1. Existing CPU usage solutions for active networks or mobile agents platforms

In order to prevent malicious or erroneous active packets from consuming too much CPU time, most execution environments implement specific mechanisms. In this section, we discuss the most common mechanisms.

Limit fixed by the packet. Some execution environments, such as ANTS [20], assign a time-to-live (TTL) to each active packet. An active node decreases this TTL as a packet transits the node, or whenever the node creates a new packet. In this way, each active packet can only consume resources on a limited number of nodes, but individual nodes receive no protection. The current recommendation for IP is 64 hops [16], which is supposed to roughly correspond to the maximum diameter of the Internet. This value might prove large enough for an active packet that propagates a configuration from node to node between two videoconferencing machines. But if the active packet creates numerous additional packets (to which it delegates a part of its own TTL), then the assigned TTL could prove insufficient. And it is usually difficult to predict how many new packets will be generated since these predictions might depend on network parameters, such as congestion and topology, which can rarely be known in advance. This TTL mechanism could contribute to protect individual nodes if the TTL is given in CPU time units instead of hop count. But the problem remains to choose the initial value for the TTL.

In the related context of mobile agents, Huber and Toutain [9] propose to enable packets that did not complete their mission to request additional credits. The decision to grant more credit would be taken by the originating node for its packets, or by the generating packet for packets created while moving among nodes. The decision must be made after examining a mission report included with the request. The proposed solution was never implemented, perhaps because the reports proved difficult to generate and evaluate.

Limit fixed by the node. In some execution environments (e.g., ANTS), a node limits the amount of CPU time any one packet can use. This solution protects the node but does not allow optimal management of resources. For instance, imagine that a node limits each packet to 10 CPU time units. Suppose that a packet requiring 11 CPU time units arrives when the node is not busy. In this case, the node will stop the execution of the packet just before it completes.

Restricted language. The SNAP language [13] is designed with limited expressiveness so that a SNAP program uses CPU in linear proportion to the packet's length. While this approach provides effective control of resource usage, it could prove too restrictive for expressing arbitrary processing in active applications. For instance, only forward branches are allowed; as a result, if repetitive processing is required, the packet must be resent repeatedly in loop-back mode until the task is completed.

Market-based approach. Yamamoto and Leduc [21] describe a model for trading resources inside an active network node, based on the interaction between a "reactive user agent" included in the capsule, and resource manager agents that reside in the network nodes. The manager agents propose resources (such as link bandwidth, memory, or CPU cycles) to the user agents at a price that varies as a function of the demand for the resource (the higher the demand, the higher the price). Capsules carry a budget that allows them to afford resources in the active nodes. Based on the posted price of the resources and on its remaining credit, the user agent of a capsule makes decisions about the processing to apply to the capsule. For instance, if the CPU is in high demand and thus expensive to use, then a capsule may decide to apply a simple compression algorithm to its data, instead of a more efficient but more costly algorithm, which the capsule would have applied if the resource were more affordable. This approach, which might prove appropriate for mobile agent platforms, could increase the capsule complexity too much to be used efficiently in active networks.

Our critique. The two most common approaches to resource control in active networks apply a fixed limit on the CPU time allocated to an active packet. In one approach, each node applies its own limit to each packet, while in the other approach each packet carries its own limit, a limit that might prove insufficient on some nodes a packet encounters and overly generous on other nodes. Neither approach provides a means to establish an appropriate limit for a variety of active packets, executing on a variety of nodes. Our research aims to solve this problem, while at the same time we aim to develop a solution that would not reduce the expressiveness of a capsule, nor make a capsule too complex.

4.2. Resource requirements need quantification

While we are unaware of any other projects aiming to quantify the CPU requirements of an AA in a heterogeneous network, we did survey several related research projects that could help us to devise an effective solution. The following sections outline and discuss some ideas we found.

Use RISC cycles? The active network architecture documents specify that a node is responsible to allocate and schedule its resources, and more particularly the CPU time. Calvert [5] emphasizes the need to quantify computing requirements of an AA in a context where these needs can vary greatly from one node to another, and suggests using RISC cycles as a unit to express computing requirements. He does not address two crucial questions. First, for a given AA, how can a programmer evaluate the number of RISC cycles required to execute a packet on a given node? Second, how can this number be converted into a meaningful unit for non-RISC machines?

Use Deus ex machina? In the AppLeS (application-level scheduling) project [3], the programmer provides information about the application that she wishes to execute on a distributed system. She must indicate for instance whether the application is more communication-oriented or computation-oriented or balanced, the type of communication (e.g., multicast or point-to-point), and the number of floating-point operations (in millions) performed on each data structure. Using this information, a scheduling program produces a schedule expected to lead to the best performance for the application. This method can lead to acceptable predictions only if the programmer is both willing and able to provide the required characteristics of the program. Discussions with software performance experts led us to think this is rarely the case.

Use combined node-program characterization? Saavedra-Barrera and colleagues [18] attempted to predict the execution time of a given program on various computers. To describe a specific computer, they used a vector to indicate the CPU time needed to execute 102 well-defined Fortran operations. In addition, they provided a means to analyze a Fortran program, reducing it to the set of well-defined operations. The program execution time can then be predicted by combining the computer model with the program model. The approach yielded good results for predicting the CPU time needed to execute one specific run of a program on different computer nodes. These results encouraged us to model platforms separately from applications; however, we need to capture multiple execution paths through each application, rather than a single path. We are pursuing a separate thread of research, not reported here, which aims to apply insights from Saavedra-Barrera to the active-network environment.

Use acyclic path models? To measure, explain, or improve program performance, a common technique is to collect profile information summarizing how many times each instruction was executed during a run. Compact and inexpensive to collect, this information can be used to identify frequently executed code portions. Unfortunately, such profiles provide no detail on the dynamic behavior of the program (for instance, these techniques do not capture and report iterations). To solve this problem a detailed execution trace must be produced, listing all instructions as they are executed. But as program runs become longer, the trace becomes larger and more difficult to manipulate. Ball and Larus [2] propose an intermediate solution: to list only loop-free paths, along with their number of occurrences. Among other things, the authors demonstrate how the use of these acyclic paths can improve the performance of branch predictors. We might be able to exploit such algorithms to efficiently capture looping behaviors; however, to collect acyclic path information we would need to instrument the program code for each application to be modeled. Given the variety of execution environments and active applications being devised by researchers, we decided to first evaluate some simpler approaches.

5. NIST model for CPU usage in active networks

In this section, we describe our model to predict an AA's CPU requirements. Then we discuss how instances of our model can be transformed to account for the capabilities of various active-network nodes.

5.1. Model and predict application requirements

Elsewhere, we define a model to represent CPU-time usage of AAs as a function of the processor cycles used in Node OS system calls, and within a specific EE between Node OS system calls [7]. Here we simply summarize.

Our AA model consists of two parts. The first part identifies the scenarios observed in an execution trace of an AA. We describe each scenario as a set of transitions between system calls, and we assign a probability to the scenario, as determined by the execution trace. The second part characterizes processor cycles required by each element of a scenario. Elements include both system calls and transitions between system calls. We define each element with a histogram to represent its CPU profile. For instance, an element might require between 5-10 processor clock cycles (pcc) with a probability of 0.2, between 10-15 pcc with a probability of 0.5 and between 15-20 pcc with a probability of 0.3.

We then use this model to predict the mean and high percentiles[1] of the CPU execution time of an AA. First, a Monte Carlo test selects a scenario, and for each element of the scenario (system calls and transitions), another Monte Carlo test selects a bin in the describing histogram. The sum gives a simulated execution time. After repeating this process enough times, the mean and high percentiles can be estimated.

Our previous results document the accuracy with which these predictions hold for a single node [7]. In general, we found standard error to range between 0 and 11 % for predicting the mean, and between 2 and 24 % for predicting high percentiles. The following section discusses our approach to adapt our predictions to provide meaningful information for various nodes. We also provide new results that demonstrate how well our adapted predictions hold when scaled across a number of nodes running the Magician EE. In later sections, we describe the use of Magician as the platform for the AA deployed in two experiments.

5.2. Transforming application models for interpretation among heterogeneous nodes

We provide a benchmark workload so each node can calibrate itself with respect to EE performance, and with respect to system-call performance [6]. From executing the benchmark, a node obtains two vectors. The EE vector gives the average user-mode pcc for the node to execute the benchmark for each EE. Similarly, the system-call vector gives the average pcc for the node to execute each Node-OS system call. We select one node as a reference, and flood its calibration vectors to all nodes in the network. To transmit an AA model between two nodes x and y, the model is subjected to a “Node-to-Reference transform”: the values describing the time spent between two system calls are dilated or contracted using the ratio T_{ref}/T_x, where T_{ref} is the average pccs required by the EE to execute the calibration workload on the reference node and T_x is the average time taken by the EE to execute the calibration workload on node x. In the same manner, we use the system-call vectors to transform the times spent in each system call. We then transmit the transformed model to node y, where the model is subjected to an inverse (the ratio is T_y/T_{ref})

[1]We use the 80^{th}, 85^{th}, 90^{th}, 95^{th}, and 99^{th} percentiles. For example, the 80^{th} percentile is the time within which 80% of the executions complete.

			Scaling with model		Scaling with processors speeds	
AA	Node X	Node Y	Mean	Avg. High Perc.	Mean	Avg. High Perc.
Ping	K	B	18	12	49	32
	R	G	21	32	74	72
	R	K	3	14	18	16
	Y	K	8	18	84	81
	G	Y	4	18	193	160
Route	K	Y	16	22	341	404
	Y	R	2	14	76	75
	K	B	6	13	13	30
	G	K	13	11	46	52
	Y	G	2	21	57	58
Audio	Y	B	11	27	85	83
	K	Y	13	14	400	399
	G	Y	9	10	80	143
	G	B	9	17	74	58
	Y	K	7	12	80	80

Table 1. % error in predictions after scaling

“Reference-to-Node transform”. The combination of these two transforms scales the pcc values within an AA model from a form meaningful on node x into a form meaningful on node y. Table 1 compares the effectiveness of our approach to scale models, against a more naïve approach that scales models based solely on differences in processor speed. The columns entitled “Avg. High Perc.” give the average among the absolute value of the error on the 80^{th}, 85^{th}, 90^{th}, 95^{th} and 99^{th} percentiles. The three presented applications (Ping, Route, and Audio) were running in the Magician EE [10] (see [7] for results about ANTS AAs). As shown in Table 1, when using our model to transform CPU-time requirements between various pairs of nodes (individual nodes are labeled K, B, R, G, and Y), the standard error for predicting the mean ranges between 2 and 21 %, and for high percentiles between 10 and 32 %. This represents an increase in error over predictions made for a single node.

6. Experiment # 1: controlling CPU usage by mobile code

In this experiment, we compared the effectiveness of our AA model against fixed allocation to control CPU usage by an AA as it traverses heterogeneous nodes in an active network. As shown in Figure 2, we constructed a four-node, heterogeneous active network, consisting of sending (200

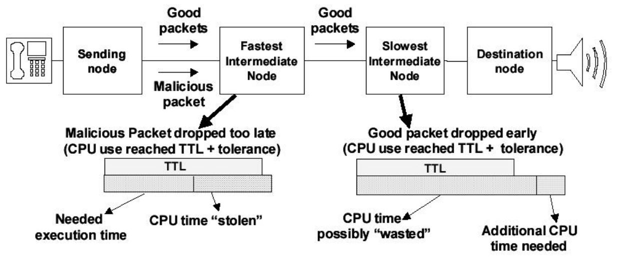
Good packets
Good packets
Sending node
Fastest Intermediate Node
Slowest Intermediate Node
Destination node
Malicious packet
Malicious Packet dropped too late (CPU use reached TTL + tolerance)
Good packet dropped early (CPU use reached TTL + tolerance)
TTL
TTL
Needed execution time
CPU time "stolen"
CPU time possibly "wasted"
Additional CPU time needed

Figure 2. Topology for experiment #1

MHz Pentium[2] Pro/64 Mbytes) and destination nodes (450 MHz Pentium II/128 Mbytes) separated by two intermediate nodes: one faster (333 MHz Pentium II/128 Mbytes) and one slower (100 MHz Pentium/64 Mbytes). All nodes included Magician [10] running on a Java^{TM} virtual machine (jdk 1.2.2) supported by Linux (release 2.2.7). We configured the experiment nodes to run an active audio application. Our application data stream generated normal active packets (Good packets in Figure 2), as well as erroneous active packets (Malicious packets in Figure 2). The erroneous packets attempt to consume as much CPU time as possible. For every five good packets the source sent a malicious packet, programmed to compute in an infinite loop. The audio application emitted a stream of 91,105 bytes, or 2278 good packets.

We conducted two experiment runs. In the first run, we assigned fixed CPU-time (TTL in Figure 2) per packet, determined by measuring the active audio application executing on the sending node. Note that this fixed time equates to a different number of pccs on each node, depending on processor speed. In the second run, we used the model discussed in Section 5 to assign CPU-time limits adjusted to account for the needs of good packets running on each of the nodes.

We expected that the malicious packets would all be stopped on the first intermediate node, thanks to the CPU control mechanism, and that less CPU time would be wasted before those malicious packets are terminated. Indeed, using our adaptive CPU-time model lowers the TTL value, and thus reduces the CPU time stolen by malicious packets in comparison with the fixed CPU-time allocation approach. Table 2 gives an analysis predicting that in this experiment our approach should lead to an average saving of 0.5 ms per packet.

Table 3 presents the results obtained from experiments. The improvement in average CPU usage of 0.52 ms per

[2]Certain commercial products or company names are identified in this report to describe our study adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the products or names identified are necessarily the best available for the purpose.

Parameter	Value
Good Packets (G)	2278
Malicious Packets (M)	379
Time-to-Live (ms) (T)	39.87
NIST Model Threshold 99th percentile (P)	36.35
Expected Improvement in CPU Utilization (I) (ms/packet) $I = ((T–P)\ (M))/(G+M)$	0.5

Table 2. Expected results from the CPU control experiment

Metric	Fixed TTL Model	NIST CPU Model	Improvement
Average CPU Utilization (ms/packet) [Fast Intermediate Node]	8.15	7.63	0.52 (6.3%)
CPU Utilization Variance (ms/packet) [Fast Intermediate Node]	0.42	0.3432	0.0768 (18.3%)
Good Packets Killed [All Nodes]	72 (3%)	10 (0.4%)	62 (2.3%)

Table 3. Experimental results from the CPU control experiment

packet is commensurate with the analytical value we computed in Table 2. Figure 3 reveals the per-packet saving by measurement interval. The per-packet saving is computed by substracting the CPU utilization using our adjusted model from the CPU utilization using a fixed TTL. During some intervals the adjusted model provides substantial per-packet savings, while in other intervals the TTL performs better. The per-interval results depend on the number of malicious packets present during each measurement interval. Beyond per-packet savings in CPU utilization, the third row of Table 3 shows also that more good packets are permitted to run to completion when we use the adjusted CPU-time model. In effect, use of a well-scaled AA model reduces the stolen and wasted CPU time in our heterogeneous active network. The same reasoning should apply in other applications, such as applets, servlets, and scripts.

Beyond controlling CPU usage, our models can also be used to predict CPU demands, when combined with the Active Virtual Network Management Prediction (AVNMP)

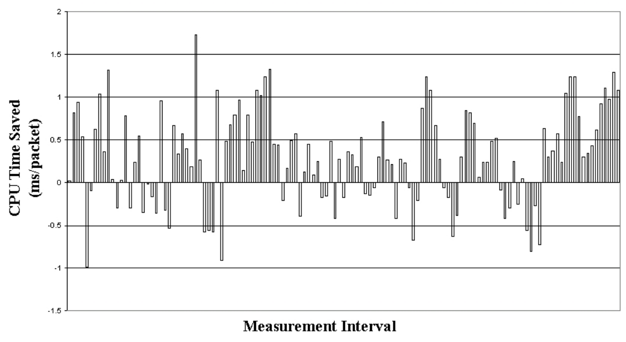

Figure 3. CPU time saved per measurement interval

system [4]. We address this next.

7. Predicting resource use with AVNMP

AVNMP constructs a shadow topology that overlays an operational network and then runs a simulation in the shadow topology to predict traffic load. Figure 4 illustrates the relationship between the operational network and the shadow, prediction-overlay network. Using Magician [10], AVNMP deploys driving processes (DP) at each source node and logical processes (LP) at each intermediate and destination node in the topology of the operational network. DPs and LPs are deployed as AAs within an active virtual-overlay network (space dimension in Figure 4). Each DP contains a model that simulates message sources, generating virtual messages that flow along links in the virtual-overlay network, which share physical links between nodes but remain logically isolated from operational traffic. As virtual messages arrive, the LP updates variables in the node's management information base (MIB) [17]. Each LP updates the future state of relevant MIB variables, providing the MIB with predicted state to complement the current and past state maintained by the operational network. After updating predicted MIB variables, the LP consults the node's routing table and forwards incoming virtual messages on to other LPs, if required.

The prediction-overlay network then generates and routes simulated network traffic that attempts to run ahead in virtual time of operational network traffic (time dimension in Figure 4). While the operational network advances in real time, the LP in the prediction-overlay network advances in virtual time, receiving virtual messages and estimating future load. Periodically, the LP compares the actual and predicted MIB values for corresponding intervals in real and virtual time. If the values agree within an error tolerance, then the simulation remains ahead of real time and continues to advance. If not, then the LP rolls virtual

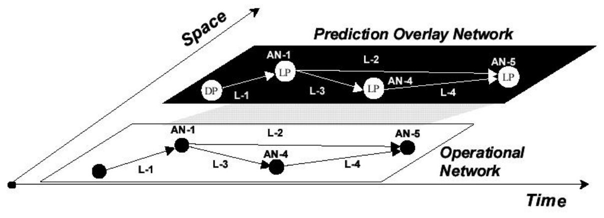

Figure 4. AVNMP as a prediction overlay network

time back to the current real time, discarding predictions for future MIB state, and then simulation resumes. AVNMP contains some special processing to cancel virtual messages that might be in transit across the prediction overlay network during a rollback, but we omit these details.

8. Experiment # 2: predicting CPU usage in a heterogeneous active network

In our second experiment, based on the same topology shown in Figure 2, we used AVNMP to inject CPU-time models for the active audio application into the prediction-overlay network in order to predict CPU usage by the AA. Figure 5 illustrates the changes made to the initial configuration. We set AVNMP to maintain a specified error tolerance between the actual and predicted CPU usage for an AA. The prediction overlay network included AVNMP deployed as an active application on each node, with a DP injected into the source node and an LP injected into the destination and each intermediate node. The DP included a message model to generate virtual message traffic and a CPU model to simulate processor use associated with each virtual message. Each LP included a CPU model to simulate processor use for each arriving virtual message.

We conducted two experiment runs. In the first run the DP and LPs predict a fixed CPU time for each virtual message on every node. In the second run, the average CPU time predicted for each virtual message differs on each node. Table 4 shows the relevant experiment parameters at each intermediate node. We assigned 7 ms per packet for the fixed CPU-time model. This figure was obtained by measuring the active audio application executing on the source node. Note that 7 ms equates to a different number of pccs on each node, depending on processor speed. When we adapt the CPU-time model to account for node differences, the model predicts that each active audio packet will take 3 ms on the fastest intermediate node and 16.5 ms on the slowest intermediate node. Our hypothesis: because an adapted model more accurately represents CPU use in an AA, as compared against a fixed-time model, AVNMP

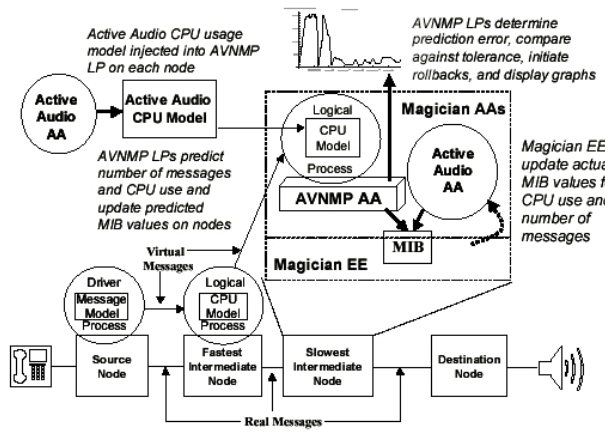
Active Audio CPU usage model injected into AVNMP LP on each node
AVNMP LPs determine prediction error, compare against tolerance, initiate rollbacks, and display graphs
Active Audio AA
Active Audio CPU Model
Logical CPU Model Process
Magician AAs
Active Audio AA
AVNMP LPs predict number of messages and CPU use and update predicted MIB values on nodes
AVNMP AA
Magician EEs update actual MIB values for CPU use and number of messages
MIB
Magician EE
Virtual Messages
Driver Message Model Process
Logical CPU Model Process
Source Node
Fastest Intermediate Node
Slowest Intermediate Node
Destination Node
Real Messages

Figure 5. Configuration for experiment # 2

	Fixed TTL Model		NIST CPU Model	
Experiment Parameter	Fastest Intermed. Node	Slowest Intermed. Node	Fastest Intermed. Node	Slowest Intermed. Node
Avg. CPU Time (ms and clock cycles)	7 2,340,750	7 693,000	3 900,000	16.5 1,633,478
Error Tolerance (± 10 %) (clock cycles)	234,075	69,300	90,000	163,347
Avg. Measurement Interval (seconds)	8.8	12.1	10.1	7

Table 4. Relevant experiment parameters for each node

should require fewer tolerance rollbacks, and provide better simulation look-ahead.

For both experiment runs we fixed the relative error tolerance at 10%, which means that AVNMP initiates tolerance rollbacks whenever the measured CPU use (averaged over 20 messages) differs from the predicted CPU use by more than 10%. This tolerance, computed relative to predicted CPU use, equates to a different number of clock cycles for each node and run. In conducting each run, the intermediate nodes periodically measured the cumulative tolerance rollbacks and the virtual time. As shown in Table 4, the average measurement interval varied on each node due to the stochastic nature of thread scheduling in Java. Table 5 and Figures 6 and 7 compare some sample results we obtained from our experiment runs. The results support our hypothesis.

	Fixed TTL Model		NIST CPU Model	
Metric	Fastest Intermed. Node	Slowest Intermed. Node	Fastest Intermed. Node	Slowest Intermed. Node
Maximum Look-ahead (seconds)	265	0	437	28
Tolerance Rollbacks	93	47	67	20

Table 5. Comparing AVNMP performance

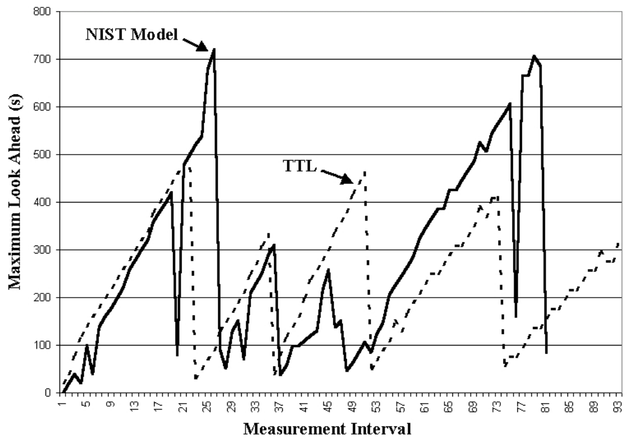

Figure 6. Maximum look ahead

9. Future work

In this section we consider some deficiencies in our current techniques to model CPU-time requirements, and we speculate about a possible approach to improve the efficiency of AVNMP. For both topics we discuss some indicated future research.

9.1. Improved models

As explained in earlier work [7], one of the main limitations of our approach is the difficulty of capturing representative behavior: our existing model assumes that all application behavior can be measured prior to injecting a model into network nodes, during a tracing phase. Unfortunately, application behaviors often reflect conditions that cannot be known before a program reaches a node. For this reason, we need to investigate solutions to overcome the fact that traces can misrepresent reality. One approach might be to allow a model to evolve as it travels through the network and gains experience. New scenarios could be added, and the probability of execution and the distribution of the execution times could be adjusted as the application experiences more execution paths. We also need to consider models that

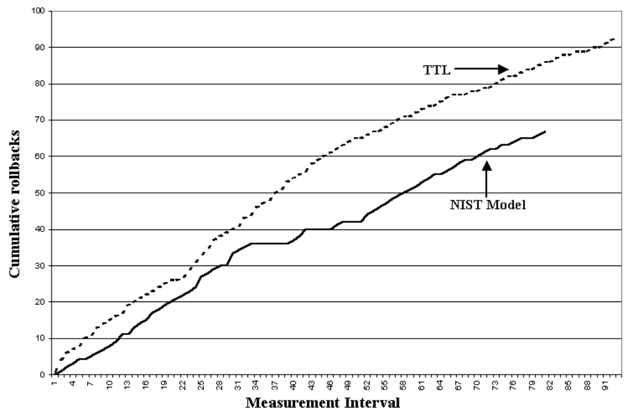

Figure 7. Cumulative rollbacks

can be parameterized based on conditions at a node. For example, to solve the problem of a loop executed an unpredictable number of times, we could design a holes-model: a complete model except for some parameters that would be included on arrival at the node, where local conditions are known.

Second, we might be unable to find a single model that will fit all applications. But using active network technology it is possible to deploy new predictors, or to extend existing predictors to process a new model introduced for a new class of applications. Further, a given model might be used in different ways to derive predictions. For instance, predictions could result from analytical computation or from simulation. Perhaps one predictor gives better results than another under certain conditions. If so, then it could prove useful to continuously evaluate which of the available co-existing models or prediction systems is the most accurate. In this way, good predictors can be reinforced, and bad predictors can be de-emphasized, and the value of predictors can be assessed independently in time and space.

The third issue to be resolved involves error characterization. Before taking decisions based on predictions from CPU-time models, an operating system must consider the possible range of prediction error. We have yet to rigorously characterize the error properties of our models. The ability of AVNMP to maintain predictions within a specified error bounds offsets this weakness in the prediction application discussed in this paper.

9.2. Improved AVNMP performance

With regard to AVNMP, we demonstrated the ability to make predictions of message load and CPU usage in a rather small network. We have yet to investigate how shadow simulations might be scaled to larger networks with thousands of MIB variables at each node. Our current system emulates real applications running in a logically isolated prediction-overlay network, which shares physical resources with the operational network. This approach will at minimum double the physical resources required from the operational network. We might be able to discover more efficient techniques to simulate future state. Such efficiency improvements, which (as detailed below) are being explored in light of recent advances in the application of Kolmogorov complexity theory [12], could prove crucial when we attempt to simultaneously predict alternative future network states. If we can achieve this goal, then AVNMP might be used to estimate multiple future states in a network, perhaps even assigning a probability to each state. Given such capability, a network manager could simultaneously explore multiple what-if scenarios and could initiate network reconfigurations based on the most likely or most critical expected outcomes. But first we must address our performance concerns with AVNMP.

To lower resource comsumption by AVNMP, we can consider application of the Minimum Data Length (MDL) [8] estimate for Kolmogorov Complexity[3]. Resource consumption by AVNMP is tied directly to accuracy: higher accuracy costs more in terms of bandwidth utilization, associated with simulation rollbacks and the concomitant transmission of anti-messages. Despite this relationship, potential exists to nearly reach the theoretical minimum amount of bandwidth to achieve maximal model accuracy. This possibility arises because AVNMP consists of many small, distributed models (each a description of a theory) that work together in an optimistic, distributed manner via message passing (data). Each AVNMP model can be transferred, using Active Networks, as a Streptichron [11], which is any message that contains an executable model in addition to data. Using Streptichrons, the optimal mix of data and model can be transmitted so as to closely approximate the minimum MDL. Achieving maximal model accuracy at minimal bandwidth would provide the best accuracy at the least cost in resource consumption.

Other possibilities exist to exploit Kolmogorov Complexity to improve AVNMP performance. For example, one can apply the MDL technique to the rollback frequency of all the AVNMP enhanced nodes in a network. A low rollback complexity (which suggests a high compressibility in the observed data) would indicate patterns in the rollback behavior that could be corrected relatively easily by tuning AVNMP parameters. High complexity (low compressibility) would indicate a lack of computable patterns, and would suggest that little performance improvement could be achieved by simply tuning parameters. Thus, we hy-

[3]The Minimum Data Length (MDL) estimate for Kolmogorov Complexity proposes that the best measure for complexity of an information unit minimizes the sum of the length, in bits, of the description of a theory that produces the unit, and the length, in bits, of the unit encoded using the theory.

pothesize that our tuning gradient should be guided toward regions of high complexity. As our research into the study of complexity, particularly Kolmogorov Complexity, proceeds, we hope to apply insights such as this to improve AVNMP performance specifically, and to assist in automating network management generally.

10. Conclusions

We outlined the importance of accurate models for predicting CPU usage by mobile code in heterogeneous networks. We described one possible model. We showed that transforming such models among heterogeneous nodes must account for a variety of factors. We experimented with our model in two different applications: control and prediction. We demonstrated that the model provides improvements over fixed-time schemes, and we argued that the model provides significant improvements over transformation approaches that consider only the relative differences among processor speeds. Further, we discussed some of the limitations of our current work, and related future research ideas.

Acknowledgments. The work reported in this paper was founded jointly by the National Institute of Standards and Technology (NIST) and the Defense Advanced Research Projects Agency (DARPA). We thank particularly Doug Maughan, DARPA's Active Networks program manager.

References

[1] D. S. Alexander, P. B. Menage, A. D. Keromytis, W. A. Arbaugh, K. G. Anagnostakis, and J. M. Smith. The price of safety in an active network. *Journal of Computers and Networks, Special Issue on Programmable Switches and Routers*, March 2001.

[2] T. Ball and J. R. Larus. Using paths to measure, explain, and enhance program behavior. *IEEE Computer*, July 2000.

[3] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Supercomputing '96*, September 1996.

[4] S. F. Bush and A. B. Kulkarni. *Active Networks and Active Virtual Network Management Prediction: A Proactive Management Framework*. Kluwer Academic/Plenum, Boston, March 2001. ISBN 0-306-46560-4.

[5] K. L. Calvert. Architectural framework for active networks, version 1.0, draft. http://www.dcs.uky.edu/∼calvert/arch-latest.ps, July 1999.

[6] Y. Carlinet, V. Galtier, K. Mills, S. Leigh, and A. Rukhin. Calibrating an active network node. In *Proceedings of the 2nd International Workshop on Active Middleware Services*, pages 115–125, August 2000.

[7] V. Galtier, K. L. Mills, Y. Carlinet, S. D. Leigh, and A. Rukhin. Expressing meaningful processing requirements among heterogeneous nodes in an active network. In *Proceedings of the Second International Workshop on Software and Performance*, pages 20–28, September 2000.

[8] Q. Gao, M. Li, and P. M. Vitanyi. Applying mdl to learning best model granularity. arXiv:physics/0005062, May 2000.

[9] O. J. Huber and L. Toutain. Mobile agents in active networks. In *ECOOP'97 Workshop on Mobile Object Systems*, June 1997.

[10] A. Kulkarni, G. J. Minden, R. Hill, Y. Wijata, S. Sheth, H. Pindi, F. Wahhab, A. Gopinath, and A. Nagarajan. Implementation of a prototype active network. In *Proceedings OPENARCH '98*, 1998.

[11] A. B. Kulkarni and S. F. Bush. Active network management, kolmogorov complexity, and streptichrons. Technical Report 2000CRD17, GE-CRD, 2000.

[12] M. Li and P. Vitanyi. *Introduction to Kolmogorov Complexity and its Applications*. Springer-Verlag, August 1993.

[13] J. T. Moore, M. Hicks, and S. Nettles. Practical programmable packets. In *IEEE InfoCom 2001*, April 2001.

[14] L. Peterson. Nodeos interface specification. http://www.dcs.uky.edu/∼calvert/nodeos-latest.ps, January 2000.

[15] X. Qie, A. Bavier, L. Peterson, and S. Karlin. Scheduling computations on a software-based router. In *Sigmetrics 2001*, June 2001.

[16] J. Reynolds and J. Postel. Rfc 1700 assigned numbers, October 1994.

[17] M. T. Rose. *The Simple Book: An Introduction to the Management of TCP/IP Based Internets*. Prentice-Hall, 1991.

[18] R. H. Saavedra-Barrera, A. J. Smith, and E. Miya. Machine characterization based on an abstract high-level language machine. *IEEE Transactions on Computers*, December 1989.

[19] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active networks research. *IEEE Communications Magazine*, 35(1):80–86, 1997.

[20] D. Wetherall, J. Guttag, and D. Tennehouse. Ants: Network services without the red tape. *IEEE Computer*, pages 42–48, April 1999.

[21] L. Yamamoto and G. Leduc. An agent-inspired active network resource trading model applied to congestion control. In *MATA 2000*, pages 151–169, September 2000.